

CS 61: Database Systems

MongoDB Schema Design

Agenda

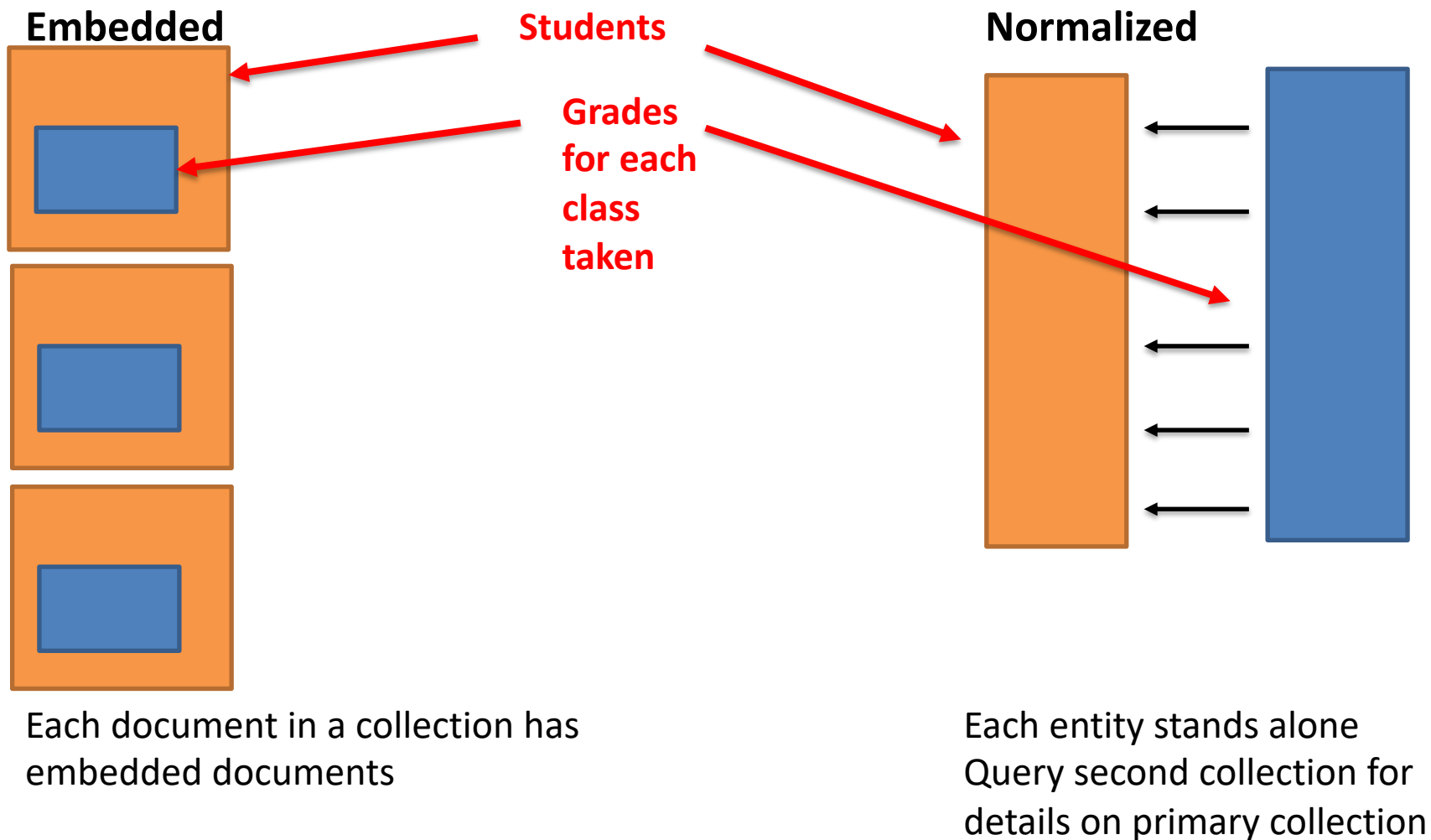


1. Data relationships

2. Accessing embedded documents

The big schema design question is whether to embed documents or normalize

Embedded vs normalized



Embedded data model moves all fields into one document

```
> db.Students.find().pretty()
```

```
{
  "_id" : ObjectId("ABC"),
  "name" : "Alice",
  "year" : 20,
  "GPA" : 3.5,
  "grades" : [
    {
      "class" : "CS1",
      "grade" : "A"
    },
    {
      "class" : "CS10",
      "grade" : "A-"
    }
  ]
}
```

Also known as a denormalized data model

Allows applications to store related pieces of information in the same database record

Improves read performance

Result is fewer database queries and updates (no joins needed)

Writes to documents are atomic

Use when:

- Have a “contains” or “has” relationship between entities
- 1:M relationship when $M \lesssim 1000$ and when many side will always appear with one side (not stand alone)
- Document must be < 16 MB in size

Normalized data model references other documents, like a relational database

Normalized data model

Like normalized tables in RDBMS

Students collection

```
> db.Students1.find().pretty()
```

```
{
  "_id" : ObjectId("ABC"),
  "name" : "Alice",
  "year" : 20
}
```

Referential integrity is not enforced

Use when:

- Embedding would result in duplication of data, but would not improve read performance enough to outweigh duplication
- To represent complex M:N relationships
- To model large hierarchical datasets

Grades collection

```
> db.Grades.find().pretty()
```

```
{
  "_id" : ObjectId("123"),
  "student_id" : ObjectId("ABC"),
  "class" : "CS1",
  "grade" : "A"
}
```

```
{
  "_id" : ObjectId("124"),
  "student_id" : ObjectId("ABC"),
  "class" : "CS10",
  "grade" : "A-"
}
```

Writes are not atomic across collections, but MongoDB has transactions

1:1 relationships often suggest using embedded documents

1:1 relationships

Normalized

// patron collection

```
{
  _id: "joe",
  name: "Joe Bookreader"
}
```

// address collection

```
{
  patron_id: "joe", //patron
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}
```

Embedded

```
{
  _id: "joe",
  name: "Joe Bookreader",
  address: {
    street: "123 Fake Street",
    city: "Faketon",
    state: "MA",
    zip: "12345"
  }
}
```

- **Embed address into patron document**
- **Now one database read gets both patron and address info vs. two reads for normalized approach**
- **Embedding is the preferred approach**

1:1 relationship counter-example is the subset problem, use normalized approach

1:1 relationship subset problem

```
{
  "_id": 1,
  "title": "The Arrival of a Train",
  "year": 1896,
  "plot": "A train is seen pulling into a station"
  "fullplot": "A group of people are standing in a straight line along..."
  "type": "movie",
  "directors": [ "Auguste Lumière", "Louis Lumière" ],
  "imdb": {
    "rating": 7.3, "votes": 5043, "id": 12
  },
  "countries": [ "France" ],
  "genres": [ "Documentary", "Short" ],
  "tomatoes": {
    "viewer": {
      "rating": 3.7, "numReviews": 59
    }
  }
}
```

If you normally only need summary data about a movie, then having plot and fullplot means more disk block reads

Create separate collection for movie details

Leave summary fields in main collection

Only read details when needed

1:M relationships: embed documents if number of embedded document is small

1:M embedded relationships

Normalized

// patron collection

```
{ _id: "joe",  
  name: "Joe Bookreader" }
```

// address collection

```
{ patron_id: "joe", //patron  
  street: "123 Fake Street",  
  city: "Faketon",  
  state: "MA",  
  zip: "12345" }
```

```
{ patron_id: "joe", //patron  
  street: "1 Some Other Street",  
  city: "Boston",  
  state: "MA",  
  zip: "12345" }
```

Embedded

**Max document
size is 16MB**

```
{ "_id": "joe",  
  "name": "Joe Bookreader",  
  "addresses": [  
    { "street": "123 Fake Street",  
      "city": "Faketon",  
      "state": "MA",  
      "zip": "12345" },  
    { "street": "1 Some Other Street",  
      "city": "Boston",  
      "state": "MA",  
      "zip": "12345" }  
  ]  
}
```

- All addresses read in with one read of document
- No need for a JOIN operation to get addresses
- Subset problem applies here too
- Use if address does not need to stand alone

1:M relationships: use normalized references to avoid duplication

1:M normalized relationships

//books collection

```
{ title: "MongoDB: The Definitive Guide",  
  author: [ "Kristina Chodorow", "Mike Dirolf"],  
  published_date: ISODate("2010-09-24"),  
  pages: 216,  
  language: "English",  
  publisher: { name: "O'Reilly Media",  
               founded: 1980, location: "CA" }  
}
```

```
{ title: "50 Tips and Tricks for MongoDB ",  
  author: "Kristina Chodorow",  
  published_date: ISODate("2011-05-06"),  
  pages: 68,  
  language: "English",  
  publisher: { name: "O'Reilly Media",  
               founded: 1980, location: "CA" }  
}
```

//publisher collection

```
{ _id: "oreilly",  
  name: "O'Reilly Media",  
  founded: 1980,  
  location: "CA" }
```

//books collection

```
{ _id: 123456789,  
  title: "MongoDB: The Definitive Guide",  
  author: [ "Kristina Chodorow", "Mike Dirolf"],  
  published_date: ISODate("2010-09-24"),  
  pages: 216,  
  language: "English",  
  publisher_id: "oreilly" }  
{ _id: 234567890,  
  title: "50 Tips and Tricks for MongoDB ",  
  author: "Kristina Chodorow",  
  published_date: ISODate("2011-05-06"),  
  pages: 68,  
  language: "English",  
  publisher_id: "oreilly" }
```

M:N relationships can be easily implemented with two-way referencing

M:N

```
db.person.findOne()  
{ _id: ObjectId("ABC"), name: "Alice",  
  tasks [ // Alice is assigned three tasks  
    ObjectId("123"), //write lesson plan below  
    ObjectId("124"), //another task  
    ObjectId("125") //Alice's third task  
  ]  
}
```

```
db.tasks.findOne()  
{ _id: ObjectId("123"), description: "Write lesson plan", due_date: ISODate("2014-04-01"),  
  assigned: [ObjectId("ABC") // Reference to Alice  
    ObjectId("DEF") //Reference to another person assigned to this task  
  ]  
}
```

Advantage:

- Easy to find who is assigned to tasks, and which tasks a person is assigned

Disadvantage:

- If person added to removed from task, must update two tables

Two collections

One *person* is assigned many tasks

One *task* is assigned to many people

Create array of references

- Person to task
- Task to person

Sometimes it is useful to denormalize

M:N

```
db.person.findOne()
{ _id: ObjectId("ABC"), name: "Alice",
  tasks [ // Alice is assigned three tasks
    ObjectId("123"), //write lesson plan below
    ObjectId("124"), //another task
    ObjectId("125") //still another task
  ]
}
```

```
db.tasks.findOne()
{ _id: ObjectId("123"), description: "Write lesson plan", due_date: ISODate("2014-04-01"),
  assigned: [{person _id: ObjectId("ABC"), name: "Alice"}, // now have Alice's name
             {person_id: ObjectId("DEF"), name: "Bob"} //also have Bob's name
  ]
}
```

Advantage:

- No need to lookup people's name when finding tasks

Disadvantage:

- If Alice's name changes, must update person collection and all entries in task collection

Two collections

One *person* is assigned many tasks

One *task* is assigned to many people

**Denormalize to include person's name
in tasks collection of assigned people**

**Now do not need to look up the
names of people assigned to tasks**

**Use denormalization if many
more reads than writes**

**Do not denormalize something
that changes frequently!**

Embedding vs. normalization rules of thumb

Advice from William Zola, MongoDB Lead Technical Support Engineer

1. Favor embedding unless there is a compelling reason not to
2. The need to access an object on its own is a compelling reason not to embed
3. Arrays should not grow unbounded:
 - If there are more than a couple hundred documents on the many side, don't embed them
 - If there are more than a few thousand on the many side, don't use an array of ObjectId references
 - High-cardinality arrays are a compelling reason not to embed
4. Don't be afraid of application-level joins: if you index correctly and use the projection specifier, then application-level joins are barely more expensive than server-side joins in a relational database
5. Consider the read/write ratio when denormalizing: a field that is mostly read and only seldomly updated is a good candidate for denormalization
6. Your design should depend on how your application accesses data!

Agenda

1. Data relationships

 2. Accessing embedded documents

Add new items to array using \$push

```
> db.Students.find().pretty()  
{  
  "_id" : ObjectId("ABC"),  
  "name" : "Alice",  
  "year" : 20,  
  "GPA" : 3.5,  
  "grades" : [  
    {  
      "class" : "CS1",  
      "grade" : "A"  
    },  
    {  
      "class" : "CS10",  
      "grade" : "A-"  
    }  
  ]  
}
```

Add new grade for Alice

```
db.Students.update(  
  {name:"Alice"},  
  {$push:  
    {  
      grades:{  
        class:"CS61",  
        grade:"A"  
      }  
    }  
  }  
)
```

Find document to update

Add entry to grades array using \$push

Add new items to array using \$push

```
> db.Students.find().pretty()  
{  
  "_id" : ObjectId("ABC"),  
  "name" : "Alice",  
  "year" : 20,  
  "GPA" : 3.5,  
  "grades" : [  
    {  
      "class" : "CS1",  
      "grade" : "A"  
    },  
    {  
      "class" : "CS10",  
      "grade" : "A-"  
    },  
    {  
      "class" : "CS61",  
      "grade" : "A"  
    }  
  ]  
}
```

Add new grade for Alice

```
db.Students.update(  
  {name:"Alice"},  
  {$push:  
    {  
      grades:{  
        class:"CS61",  
        grade:"A"  
      }  
    }  
  }  
)
```

Find document to update

Add entry to grades array using \$push

Grade for CS61 added

Use \$pull to remove item from array

Access embedded document using “dot notation”


```
> db.Students.find().pretty()  
{  
  "_id" : ObjectId("ABC"),  
  "name" : "Alice",  
  "year" : 20,  
  "GPA" : 3.5,  
  "grades" : [  
    {  
      "class" : "CS1",  
      "grade" : "A"  
    },  
    {  
      "class" : "CS10",  
      "grade" : "A-"  
    },  
    {  
      "class" : "CS61",  
      "grade" : "A"  
    }  
  ]  
}
```

Find students who got an A in CS61

```
db.Students.find(  
  {  
    "grades.class":"CS61",  
    "grades.grade":"A"  
  }  
)
```

// Returns Alice document

**Reference fields in
grades array with
dot notation**



Practice

1. Design a MongoDB database for restaurant inspections from our running example

- Each restaurant has:
 - Name, Boro, Cuisine
- Each restaurant can be inspected many times, each inspection has
 - Date, Score, zero or more violations
- Each violation has
 - Code (e.g., 06N), Description (e.g., “Improperly cleaned food surface”)

2. Insert several restaurants into your database

- Name: Morris Park Bake Shop, boro: Bronx, Cuisine: Bakery
 - Inspected on 5/5/2020, score 10, violations 06N, 08B
 - Inspected on 4/4/2019, score 12, violations 12C
- Name: Tim’s Tasty Treats, boro: Manhattan, Cuisine: Fruits/Vegetables
 - Inspected on 3/3/2020, score 5, violations none
 - Inspected on 2/2/2019, score 7, violations 08B

3. Query your database

- Find all bakeries
- Find just the names of all restaurants inspected after 5/1/2020
- Find all inspections that had a violation code of 08B

